CMPU 378: Computer Graphics
Assignment 2: Drawing Program

Overview

Design and implement a program for drawing two-dimensional pictures composed of lines, rectangles, circles and polygons. The program will use a scene graph as its primary data structure for organizing, transforming and rendering the scene. The program will include the following facilities: (1) Choosing the type of shape to draw; (2) Choosing the color of a shape to be drawn; (3) Drawing the designated shape and color with the mouse; (4) Using the mouse to click-select a node in the scene graph; (5) Translating, rotating, shearing or scaling selected nodes in the scene graph; (6) Creating new scene graph nodes that represent groups of existing nodes; (7) Changing the parent of one or more nodes in the scene graph.

I have provided you with the shell of the program you will write, in the form of MS Visual Studio project. The project includes a collection of header and implementation files; a complete implementation of the user interface; and declarations of the public interfaces of the classes and functions that you need to write. These classes include the following:

- **`TransformNode`**: Class representing nodes in the scene graph.
- **`ShapeNode`**: Abstract class representing shapes that can be attached to transform nodes in the scene graph.
- Subclasses of **`ShapeNode`**: **`Line`**, **`Rectangle`**, **`Circle`** and **`Polygon`**.
- **`Vector`**: Class representing 2D points in homogeneous coordinates.
- **`Matrix`**: Class implementing 2D transformations in homogeneous coordinates.
- **`TransformStack`**: Class for maintaining a stack of transformation matrices.
- **`Graphics Package`**: Routines for drawing shapes (lines, rectangles, circles and polygons) that are stored in local coordinates but must be drawn in window/world coordinates.

Your task is to complete the declarations of these classes (by declaring private data structures and helper functions) and to implement the declared public and private functions. All of the public functions you must implement are described in the appendix to this assignment document. I have included the simple **Scene-Graph** program in the files for this project. You may want to refer to this simple scene graph implementation as a guide to your work.

Planning and Carrying-Out Your Work

I recommend that you implement the program in stages, testing each stage as you go along. You should begin by reading over the files **`drawing.h`** and **`drawing.cpp`**, which contain the implementation of the user interface. Then implement your program in the following stages:

1. Implement trivial versions of **`TransformNode`**, **`ShapeNode`**; a single shape class, say the **`Line`** class, and the primitive **`drawLine`** function. Now you can draw a line, construct a **`Line`** object, insert it under the root of the scene graph and display the scene graph – doing all of this in window/world coordinates. In order to test your first stage of implementation, you will need to comment out the calls to un-implemented functions that appear in the user interface.

2. Implement enough of the **`Vector`** and **`Matrix`** classes so that you can construct identity matrices, translation matrices and inverse matrices; multiply vectors by matrices; and multiply matrices by matrices. Implement the **`translate function`** of the **`TransformNode`** class. Modify the **`draw`** function of the **`TransformNode`** class to properly manipulate the **`TransformStack`** and the OpenGL name stack. Modify the **`drawLine`** function to use the transform stack so that lines stored in local coordinates are drawn in world coordinates Finally, implement the **`TransformStack`** class itself. In order to test this stage of implementation, you will need to uncomment the user interface code for selecting scene graph nodes and applying translations to selected scene graph nodes.

3. Complete your implementation of the **`Vector`** and **`Matrix`** classes, so that you can construct matrices for rotation, shearing and scaling. Implement the **`rotate`**, **`scale`** and **`shear`** functions of the **`TransformNode`** class. Implement the **`Rectangle`**, **`Circle`** and **`Polygon`** shape classes, and the corresponding primitive drawing functions **`drawRectangle`**, **`drawCircle`** and **`drawPolygon`**. Uncomment enough of the user interface, so that you can test each type of transformation on each type of shape.

4. Implement the remaining functions in the **`TransformNode`** and **`ShapeNode`** classes, and the remaining portions of the graphics package interface. Uncomment the rest of the user interface. Test all of the remaining operations available from the user menu, including operations for parenting, grouping, copying and deleting objects. Test your program as thoroughly as you can.

Appendix

Transform Node Public Interface

`TransformNode(TransformNode* p)`
Initialize this transform node to have parent p, shape node NULL, and transform identity.

`TransformNode(TransformNode* p, ShapeNode* s, Matrix* t)`
Initialize this transform to have parent p, shape node s and transform t.

`~TransformNode()`
Arrange that deleting this transform deletes all its descendants.

`void translate(double dX, double dY)`
Update this transform node to translate by (dx,dy) in world coordinates.

`void rotate(double theta)`
Update this transform node to rotate by theta around the origin in world coordinates.

`void shear(double sXY, double sYX)`
Update this transform node to apply shear (sXY,sYX) in world coordinates.

`void scale(double sX, double sY)`
Update this transform node to apply scale (sX,sY) in world coordinates.

`virtual void draw(bool dH) const`
Draw the portion of the scene represented by this transform node. If dH is true, display a square indicating the origin of the coordinate system of this node.

`TransformNode* getParent() const`
Return the parent of this transform node.

`void setParent(TransformNode* p)`
Set p to be the parent of this transform node.

`void changeParent(TransformNode* newParent)`
Make nP the new parent of this transform node, taking care that the local coordinate system of this node remains unchanged.

`void groupObjects(set<TransformNode*>& groupMembers)`
Construct a new transform node under this object. Make groupMembers children of the new transform node. Assume all members are presently children of this node.

`Matrix* getTransform() const`
Return the matrix representing the transform associated with this node.

`virtual TransformNode* clone() const`
Return a copy of this transform node, its shape node (if any) and its descendant transform nodes. The copy should have a NULL parent.

`void addChild(TransformNode* c);`
Add c to the collection of children of this transform node.

`void removeChild(TransformNode* c);`
Remove child c from the collection of children of this transform node.

`TransformNode* firstChild() const;`
Return the first child in the collection of children of this transform node.

`TransformNode* lastChild() const;`
Return the last child in the collection of children of this transform node.

`TransformNode* nextChild(TransformNode* c) const`
Return the child next after c in the collection of children of this transform node.

`TransformNode* previousChild(TransformNode* c) const`
Return the child previous before c in the collection of children of this transform node.

`void select();`
Mark this transform node as selected so that its subtree is drawn in a special highlight color.

`void deSelect();`
Mark this transform node as not selected, so that its subtree is drawn in its normal color.

`static TransformNode* nodeLookup(int identifier);`
Return the transform node associated with identifier.

Shape Node Public Interface

`ShapeNode(colorType c)`
Initialize this shape node sub-object with color c and NULL transform node.

`virtual void draw() const = 0`
Draw this shape node in the local coordinate system of its transform node. You will need to implement this function in classes derived from ShapeNode.

`void setTransformNode(TransformNode* tN)`
Set tN to be the transform node of this shape node.

`TransformNode* getTransformNode();`
Return the transform node of this shape node. (Note: This feature of the interface is not used in the drawing program application.)

`virtual ShapeNode* clone() const = 0`
Return a copy of this shape node, setting the transform node of the copy to NULL. You will need to implement this function in classes derived from ShapeNode.

Line, Rectangle, Circle and Polygon Public Interfaces

`Line(double x0, double y0, double x1, double y1, colorType c)`
Initialize line object of color c connecting (x0,y0) and (x1,y1) in world coordinates.

`Rectangle(double x0,double y0,double x1,double y1,colorType c)`
Initialize rectangle of color c with opposite corners (x0,y0) and (x1,y1) in world coordinates.

`Circle(double cX, double cY, double r, colorType c)`
Initialize a circle of color c with center (cX,cY) and radius r in world coordinates.

`Polygon(const list<Vector*>& vs, colorType c)`
Initialize a polygon of color c with vertices vs in world coordinates.

Vector Public Interface

`Vector()`
Initialize a zero vector.

`Vector(const Vector& oldVector)`
Initialize this vector to be a copy of oldVector.

`Vector(const double x, const double y)`
Initialize this vector to represent the point (x,y).

`~Vector()`
Destructor for vectors.

`double& operator[](int i) const`
Return a reference to the ith element of this vector.

Matrix Public Interface

`Matrix()`
Initialize this to an identity matrix.

`Matrix(const Matrix& oldMatrix)`
Initialize this matrix to be a copy of oldMatrix.

`~Matrix();`
Destructor for matrices.

`Matrix* multiply(const Matrix* otherMatrix) const`
Allocate and return a matrix representing the product of this matrix and otherMatrix.

`Vector* multiply(const Vector* theVector) const`

Allocate and return a vector representing the produce of this matrix and theVector.

`double* operator[](int i) const`
Return a pointer to an array of doubles representing the ith row of this matrix.

`static Matrix* translation(double deltaX, double deltaY)`
Allocate and return a matrix implementing a translation (deltaX,deltaY).

`static Matrix* rotation(double theta)`
Allocate and return a matrix implementing a rotation by theta (in radians).

`static Matrix* shearing(double shearXY, double shearYX)`
Allocate and return a matrix implementing a shearing (shearXY,shearYX).

`static Matrix* scaling(double scaleX, double scaleY)`
Allocate and return a matrix implementing a scaling (scaleX,scaleY)

`Matrix* getInverse() const`
Allocate and return the inverse of this matrix.

TransformStack Public Interface

`TransformStack()`
Initialize this transform stack to hold only an identity matrix.
`void push(Matrix* transform)`
Form the matrix product top()*transform and push it onto the stack.
`void pop()`
Pop this transform stack.
`Matrix* top()`
Return the matrix at the top of this transform stack.

Graphics Package Public Interface

`void drawLine(double x0, double y0, double x1, double y1)`
Consider the line that connects point (x0,y0) and point (x1,y1) given in local coordinates. Draw the line in world coordinates.
`void drawRectangle(double x0, double y0, double x1, double y1)`
Consider the rectangle with opposite corners (x0,y0) and (x1,y1) given in local coordinates, whose sides are parallel to the local coordinate axies. Draw the rectangle in world coordinates.
`void drawCircle(double x0, double y0, double x1, double y1)`
Consider the circle with radius running from (x0,y0) and (x1,y1) local coordinates. Draw the circle in world coordinates.
`void drawCircle(double xC, double yC, double r)`
Consider the circle with center (xC,yC) and radius r in local coordinates. Draw the circle in world coordinates.
`void drawPolygon(const list<Vector*>& vs, bool c)`

Consider the polygon with sides connecting successive points in vs, given in local coordinates. Draw the polygon in world coordinates. Close close the polygon if c==true. Otherwise, omit the side from the last point in vs back to the first point in vs.